ALMA MATER STUDIORUM - UNIVERSITY OF BOLOGNA
CAMPUS OF CESENA
SCHOOL OF ENGINEERING AND ARCHITECTURE

MASTER'S DEGREE IN COMPUTER SCIENCE AND ENGINEERING

Game Engines and MAS: BDI & Artifacts in Unity

Author:
Nicola POLI

Supervisor:
Prof. Andrea OMICINI

Cosupervisor:
Dr. Stefano MARIANI

Examiner:
Dr. Silvia MIRRI

Thesis in Autonomous Systems

Session: III

Academic Year: 2016/2017

February 15, 2018

Abstract

Italiano: In questa tesi vedremo un breve sunto riguardo lo stato dei Sistemi Multi-Agente e andremo ad analizzare le limitazioni che attualmente ne impediscono l'utilizzo ai programmatori di videogiochi. Dopodiché, andremo a proporre un nuovo linguaggio BDI, basato su Prolog e inspirato a Jason, che, grazie all'interprete Prolog sviluppato da I. Horswill, darà la possibilità al programmatore di videogiochi di esprimere comportamenti dichiarativi di alto livello per agenti autonomi all'interno del game engine Unity. Andremo anche a proporre una versione di Artefatto per la modellazione dell'ambiente in una scena Unity e un layer di comunicazione che agenti e artefatti possano utilizzare per interagire tra loro. Infine presenteremo un caso di studio per sottolineare i benefici che questo sistema fornisce.

English: In this thesis we will give a brief summary about the current state of Multi-Agent Systems and address some of the problems that currently forbid game programmers to use them in their work. Afterwards, we will propose a new BDI language, based on Prolog and inspired by Jason, which, thanks to the Prolog interpreter developed by I. Horswill, will give game programmers the ability to define high-level declarative behaviours for autonomous agents in the Unity game engine. We will also propose a version of Artifact for modelling the environment in a Unity scene and a communication layer which agents and artifacts can use to interact with each other. Finally, we will present a case study to underline the benefits which this system provides.

Web Materials

Full source code of the system developed in this thesis (including the case study) is available at:
https://github.com/conner985/UnityLogic

A demonstrative video of the case study is available at:
https://www.youtube.com/watch?v=BMHiZImVC3A

Acknowledgements

To my family and friends who accompanied and supported me through all these years.

To my supervisor Andrea Omicini and cosupervisor Stefano Mariani for the knowledge and passion they shared with me during many incredible projects.

Finally, a special thanks to Ian Douglas Horswill (Northwestern University) for all the interest shown in this work without which I wouldn't have been able to accomplish so much.

Contents

List of Figures

List of Tables

"When you have two competing theories that make exactly the same predictions, the simpler one is the better."

— Ockham's Razor

Chapter 1

Introduction

Nowadays, most of the frameworks used for developing games are available for amateurs and professionals as well. Lowering the entry point of these engines had a significant impact on independent game developers making them able to create a game with even little knowledge about engineering and programming. Furthermore, researchers of any possible field, can exploit the functionalities that a **Game Engine** provides (e.g. physics, collisions, environment, ...) to attain visual (2D or 3D) representation of a simulated environment and have a better understanding of a particular phenomenon.

Programmers who make use of game engines usually exploit the Object Oriented Paradigm (OOP) to realize their simulations and games but, such paradigm, is not suitable for the definition of complex intelligent agents.

The purpose of this thesis is to propose a different, more expressive and intuitive design approach for the definition of AI based on the notion of autonomous agents [KGR96] and integrate these new defined agents in a simulated environment, powered by the **Unity** [Uni18] game engine, in which they can perceive, reason and act, effectively porting the concept of **Multi-Agent System** (MAS) [Vla03] inside the game development pipeline.

1.1 Motivations

The most intuitive way to define a behaviour is by listing a sequence of actions that should be completed, in order to achieve some goal: if you want to open the fridge, first you need to go to the kitchen, then walk toward the fridge, reach the handle and finally pull. This kind of practical reasoning [VB91] is incompatible with the architecture of game engines because their logic is frame based and usually this means that with complex and long behaviours one should exploit different approaches like Finite State Machines — which, however, ruin the logical thinking that we (humans) are more accustomed to.

There is a lack of high-level abstractions to help programmers with the definition of intelligent agents in a more human-like structure.

1.2 Goals

Unity, like many other game engines, has a frame based architecture: every game is built upon an event loop that cyclically computes every frame, which also means that programmers must adapt to this logic while designing a game. For

complex simulations this kind of logic is less than ideal.

Exploiting the agent paradigm, in particular developing BDI agents, I aim to achieve a new layer of abstraction that would allow programmers to declare high-level behaviours, with ease, without any need of explicit synchronization with the event loop.

To accomplish this, a new declarative language and a logical reasoner need to emerge. An interpreter is also fundamental to make Unity able to understand this new language, activate the reasoning cycle and apply the results.

Agents alone cannot fully express everything in a Multi-Agent System because the environment, and their interactions with it, plays an important role in every simulation. With that in mind and inspired by the Agents and Artifacts meta-model [ORV08], I decided that a model for artifacts, that is, environmental objects was also necessary to have a more complete representation MAS simulation.

Chapter 2

State Of The Art

In this chapter we take a brief look at the most relevant contributions made in the autonomous agents field, with regard to BDI models and Multi-Agent Systems, then we see how these models have been applied to game design by both professionals and independents researchers whose works inspired, and marked the starting point of, this thesis.

2.1 Agents, Artifacts and MAS

It is possible, and quite common, to view a BDI agent as a rational system with mental attitudes [RG95], namely Beliefs, Desires and Intentions with those being respectively what the agent knows about the world, what motivates it and what it is doing to achieve its goals.

A system that comprises a certain number of agents who can (potentially) interact with each other is called a Multi-Agent System [Vla03]. Multi-Agent Systems have become more and more appealing thanks to their characteristics [Syc98]. These systems have totally decentralized control, data and computation and due to their modular nature they are also easy to maintain and scale.
To this day, many are the application proposals for Multi-Agent Systems: microgrids control [DH04], market simulations [Pra+03], automated management and analysis of SCADA [Dav+06] and so forth.
As for tools developed to design Multi-Agent Systems, the JADE platform [BCG07] and the Jason language [BHW07] had probably the most success. Jason, with regard to its syntax and reasoning cycle, will be the foundation of the agent side of the system defined in this thesis.

Agents alone cannot completely describe and model every aspect of a MAS, it is important to understand that the environment is as crucial as the agents that act in it. A way to model the environment and its components, must arise. Firstly proposed in [ORV08], the Artifact model was created to fill the gap between Agents and Environment in a Multi-Agent System. An Artifact should be used to model passive entities in the environment which agents can use, share and manipulate. CArtAgO [RVO07] has already been proposed as a framework to develop artifact-based environments.

While Jason will serve as the main reference for the agent side of this system, the concepts proposed in CArtAgO will be used as a base to define artifacts and their communication interface.

2.2 Agents and Games

Not many game programmers use agent paradigms for developing their projects but it is possible to find cases in which logic reasoning was applied, at some level, during the process of creating or testing a feature of a game.

It is important to make a distinction between Logical and Practical Reasoning. Purely Logical Reasoning is mostly used by agents designed for theorem proving but, as said in [Woo09], decision making is more in the field of Practical Reasoning and that is why, BDI languages like AgentSpeak(L) [Rao96], share some common grounds with Logic Programming but have several strong semantic differences (e.g. pure logic programs make no distinctions between goals inside a rule and those that form the head of such rule). That said, applications of Practical Reasoning in videogames are not easy to be found so, to give at least a little context to this thesis, we will analyze some of the most relevant works that employed logic programming.

Probably one of the most interesting project in this area is Versu [ES18]. Versu is an Interactive Narrative System entirely developed using modal logic. One of the creators, Evan Richards, worked as the AI lead for The Sims 3 and designed the AI for Black & White.

A good example of how logic programming can also be used for testing features in games was realized in [SST13] where Prolog-based agents have been used to test procedural generated content in the puzzle game Cut the Rope.

I. Horswill used a custom Prolog interpreted he designed for Unity interoperability, to create a mystery game [Hor14] based on intelligent agents able to inject beliefs and alter others' behaviours. As yet, this project is the closest thing to BDI-like agents employed in a game.

Even though it is possible to find some examples of games that used logical agents in different ways, most of them are independent researches and the reason why they still aren't employed in commercial games is based mainly on one factor: the lack of a solid high-level abstraction to represent intelligent agents fully compatible and integrated with game engines. Until a widely validated and efficient agent abstraction is made available for the developers, no game will ever feature autonomous agents able to exploit practical reasoning to make decisions.

Efficiency must not be overlooked: while working with games, even a single millisecond could make a difference in the overall experience. This is the reason why game programming comes with low-level abstractions (e.g. coroutines in Unity) and a lot of post-production tweaking to cut every avoidable computational load. This also means that, in complex games, code can become complicated and difficult to maintain, very fast.

Chapter 3

BDI Agents In Unity

In order to design an architecture for BDI agents, it is first necessary to understand how such agents work.

As proposed in [RG95], the reasoning cycle of an agent is composed by four main phases:

1. option generation - the agent will return a set of options based on what it knows about the world and what its desires are;
2. deliberation - the agent will select a subset of the previously selected options;
3. execution - the agent will execute, if there is an intention, an action;
4. sensing - the agent will finally update its knowledge of the world (and possibly itself).

As stated by the authors, this is an abstract architecture and, as such, it is not coupled to any kind of programming paradigm, nevertheless, I found the reasoning cycle of an agent to be very close to logical reasoning and thus the choice of using a logic paradigm as the foundation of my BDI system.

3.1 Prolog In Unity

There are several options to evaluate when trying to use a different paradigm with a framework that doesn't natively support it.

One, for instance, is to implement an interpreter or a compiler for this paradigm using the native language of the framework but, of course, this could require an immense amount of time.

Another possible solution is to implement the application logic that you need in the new paradigm, using a framework that actually supports it, and make the two frameworks communicate via sockets, exchanging data necessary to carry on the execution.

Sockets can be slow (in game development every ms matters) and can make the design of the game a lot less intuitive, moreover it forces the designer to work on different frameworks to create the game. For these reasons, the usage of sockets has been left as a last resort.

Since Unity uses Mono as a scripting backend and a .NET 4.6 equivalent (still

experimental in the 2017.2 version of Unity) as a runtime version, it is necessary to find a Prolog system that is compatible.

The first Prolog system I look into was tuProlog [DOR01] primarily because it is a light-weight version Prolog and it is compiled using the .NET framework but even if Unity manages to load the dll just fine, as soon as you try to instantiate the engine, a TypeLoadException is thrown. Since this is a known issue in the current version of Unity 2017.2 (Issue ID 911897), tuProlog was discarded and will not be usable at least until this issue is fixed.

Luckily enough, Ian Horswill implemented an interpreter for Prolog [Hor17] built to be fully compatible with Unity, it also has a lot of features to extend the interoperability of Prolog with GameObjects. In the next sections I will discuss the major pros and cons of using the UnityProlog project as the foundation of my BDI system.

3.1.1 Advantages

Using a version of Prolog natively designed to work with Unity can bring several advantages and simplifications while designing a system that tries to intertwine Logic Programming with game engine functionalities.

First of all, it gives the programmer full interoperability between the two systems - i.e. it is possible to call Prolog from C# and viceversa.

Since it is designed to be used within Unity, it comes with primitives that target GameObjects, Components and many other features of the engine, directly from Prolog.

Even if it's not a Prolog feature, I. Horswill decided to implement a version of the Exclusion Logic [Eva10] within its project that can render the update semantic of terms, in some particular cases, much more elegant and intuitive.

3.1.2 Major Limitations

UnityProlog comes with some major limitations that one cannot simply overlook and must take into account when choosing to go on with this version of Prolog, even if it's the only one that seems to work under Unity.

Being an interpreter of Prolog, it clearly won't be as fast as a compiler and this could be a problem for big MAS simulations.

It doesn't support rules with more than 10 subgoals - i.e. if you are planning to write a complex rule to define an agent behaviour, with lots of goals to check, you must break down the rule in sub-rules of no more than 10 subgoals each.

Apparently there is no high-level procedure to find all possible unifications of a variable when trying to solve a goal: the variable will be unified with only the first matching term.

3.2 BDI Agent Architecture

A BDI agent should at least have a set of **Beliefs**, **Desires** and **Plans**, plus a reasoner that is able to choose a plan according to its beliefs and desires, effectively turning the plan into an **Intention**. It should also be able to sense, from the environment in which resides, and act.
The resulting architecture is a slight modification of the one proposed in [Woo09]:

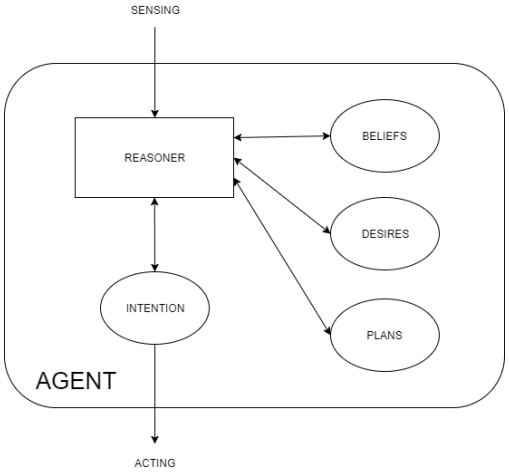

FIGURE 3.1: Architecture of a BDI Agent

The entry point of this agent is the **Reasoner** which is triggered by some external stimulus. The stimulus can alter beliefs, desires or even plans. The reasoner will filter all plans and select one that is suitable to become an intention. The intention can be backtracked, which means that if an action of the intention fails, another plan can be chosen to become the active intention.
It is also worth notice that in this general architecture, an intention can also generate stimuli that will trigger the reasoner: in this vision, an agent is a proactive entity.

Agents resulting from this architecture are almost completely platform-independent, which means that it should be possible to take the brain of the agent and, with little alterations of the reasoner, be able to place it in whatever environment we desire but, since we want to situate these agents in a Unity environment, how the interaction with its surroundings (**Sensing** and **Acting**) works, must be designed in Unity.

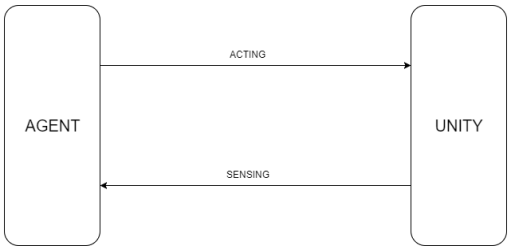

FIGURE 3.2: Agent-To-Unity Interaction

With the above considerations in mind, the brain of the agent will be realized using a Prolog interpreter while the body will be designed using the Unity framework, in particular with C# Scripts and GameObjects.

3.3 A Simple Agent

Designing a BDI agent in Unity, using a Prolog support, is a matter of deciding how much competence to give to each platform. In my overall vision, Prolog is to be used to express the behaviour of an agent in a declarative way, abstracting from the game loop as much as possible - i.e. a programmer should worry only about the logic of a behaviour, while Unity scripting in C# should be used to actually express how the actions inside a behaviour work and interact with the scene.

3.3.1 Prolog

First thing to define is what beliefs and desires are. We can think of beliefs like the knowledge of the agent, while a desire is what the agent wants to achieve. In theory, anything could be a belief or a desire so, keeping this generality a belief is a Prolog term with a prefix operator named **"belief"**. The same goes for the definition of desire with a prefix operator named **"desire"**. Both operators are defined as follows:

```
:- op(497,fx,belief).
:- op(497,fx,desire).
```

These two particular operators are not going to be interpreted, their only purpose is to syntactically separate beliefs from desires.

Regarding the definition of plans, I want them to be structured as simple and clean as possible and since some operators aren't available for redefinition on UnityProlog (e.g. !/1, :/2), I opted for this form:

```
Event && (PreConditions) => [ListOfActions].
```

Where **"&&"** and **"=>"** are operators defined as follows:

```
:- op(498,xfy,&&).
:- op(499,xfy,=>).
```

The operator && is useful to syntactically separate the event section of a plan from its preconditions. **PreConditions** is a set of Prolog predicates, wrapped in parentheses and separated by a comma from each other (e.g. (check(this), then(this), finally(this_one))), there is no formal limitation on what a precondition could be, but for the plan to be triggered, every each one of them must be verified.

Similarly, the => operator is used to syntactically separate the preconditions from

the actual actions of the plan. **ListOfActions** is a list of Prolog predicates, wrapped in square brackets and separated by commas (e.g. [do(action1), after(dothis), and(this)]), there is no formal limitation on what actions could be. For the intention to be successful every action must be verified and not fail, otherwise the intention will be backtracked and another plan, if one exists, will be chosen to become the active intention.

For simplicity, only two kind of events are taken into account at this point: addition and deletion of a desire. This choice has been made to keep the structure of the agent as clean as possible while still compatible with what UnityProlog provides: lots of operators aren't available for redefinition in UnityProlog and this forced the language structure to become very different from what a Jason programmer is used to deal with. For example in Jason is possible to define an addition triggering event of an achievement-goal using the structure "+!some(goal)" but the operator "!" is reserved in UnityProlog. With this limitation I decided to express the same semantic using the structure "add some(goal)" with "some(goal)" being a desire defined as "desire some(goal)". Both "add" and "desire", as well as "belief", are prefix operators with a single parameter and thus is not possible to combine them in a structure like "add desire some(goal)".

When someone add/remove a desire to/from the knowledge base of an agent, potentially, this action, could trigger a plan, if one is defined with such event. These two type of events are defined with prefix operators, **"add"** and **"del"**:

```
:- op(497,fx,add).
:- op(497,fx,del).
```

Events are in the form **"add D"** and **"del D"**, where D should represent a possible desire that the agent could, at some point, has in its knowledge base. To keep the plan structure more clean, there is no need to express also the "desire" operator within the Event: if we want a plan to be triggered by the addition of a desire named my_desire, the Event section of the plan would suffice to be in the form "add my_desire", while that same desire within the knowledge base will have the "desire" prefix operator "desire my_desire".

To alter the desire set, two more predicates are necessary: **add_desire/1** and **del_desire/1**. These predicates are the entry points for a programmer that want to trigger a plan: every time a desire is added into the knowledge base through add_desire/1 predicate or deleted using the del_desire/1 predicate, the reasoner will also check if, somewhere in the knowledge base, a plan that has that particular event is defined and will try to execute it. It is also possible to interrogate the knowledge base to check if some desire is present using the **check_desire/1** predicate.

```
check_desire(D) :-
    desire D.

add_desire(D) :-
    check_event(add D)
    ;
    assert(desire D).

del_desire(D) :-
    check_event(del D)
    ;
    retract(desire D).
```

What the predicate **check_event/1** does, is to search the knowledge base for a plan of the form defined above, if it finds it, it will try to verify the preconditions and to execute (verify, in logical terms) the list of actions.

```
check_event(add D) :-
    add D && C => A,
    assert(desire D),
    C,
    append(A,[del_desire(D)],Tasks),
    extract_task(Tasks).

check_event(del D) :-
    del D && C => A,
    desire D,
    retract(desire X),
    C,
    extract_task(A).
```

After an intention is completed, which means every action has been verified, the desire that was characterizing it will be automatically removed from the knowledge base, using the del_desire/1 predicate and thus triggering a possible plan with such event as precondition, because the desire is now fulfilled.

The purpose of the predicate **extract_task/1** is to verify the list of actions, one element at a time.

```
extract_task(A) :-
    (A = [], !, true)
    ;
    (A = [M|N], !, M, extract_task(N)).
```

Sometimes, even if a plan is triggered, after the addition of a desire in the knowledge base, it won't turn into an intention because not all preconditions are met (e.g. a character wants to light a fire but it has no wood) and when this happens, that desire will sit around with no possibility of being fulfilled. To fix this strong limitation I needed to plan a recheck of all unfulfilled desires. Since preconditions

are mostly checks of beliefs within the knowledge base, the best moment to plan a recheck is when a belief is either added or removed. To this purpose, three new predicates were created:

```
add_belief(B) :-
    assert(belief B),
    check_desires.

del_belief(B) :-
    retract(belief B),
    check_desires.

check_desires :-
    desire D,
    add D && C => _,
    C,
    check_event(add D).
```

It is important to notice that a programmer should always use the predicates add_belief/1 and del_belief/1 when trying to add or remove belief to/from the agent, instead of manually through the predicates assert/1 and retract/1, or if a plan is executable after the alteration, it won't be triggered.

```
check_belief(B) :-
    belief B.

check_plan(Head,Full) :-
    Head && C => A,
    Full = Head && C => A.
```

The **check_belief/1** predicate is provided as a high-level API to check the existence of a particular belief inside the knowledge base. The same goes for the predicate **check_plan/2** through which is possible to easily check if a certain plan is present, providing its head (e.g. add some(desire)).

As a final step, I wanted to give the designer a simple way to call, from within a plan, a procedure that resides in the C# script of the agent and I did so with the operator **"act"**, as in ACTion:

```
:- op(500,fx,act).
```

The "act" operator is being interpreted by extract_task/1 which will search the C# script attached to the agent for a method with the declared name, calling it with the required parameters, if such method exists:

```
act A :-
    (A = (@Ref,M,Ret), !,
        call_method(Ref,M,Ret))
    ;
    (A = (@Ref,M), !,
        call_method(Ref,M,Ret),
        Ret \= false)
    ;
    (A = (M,Ret), !,
        get_ref(Ref),
        call_method(Ref,M,Ret))
    ;
    (A = M, !,
        get_ref(Ref),
        call_method(Ref,M,Ret),
        Ret \= false).
```

This operator can be used to call a method that reside in the agent's script (3th and 4th match) or in an external script of which the reference is known (1st and 2nd match).

Summing up, this agent has a set of generic beliefs and desires, a set of plans that can be triggered by an alteration of such sets and actions that can be defined in C#, also, exploiting the backtracking of Prolog, if an intention fails, the plan will be backtracked and another one, if present, will be automatically triggered. As an example, let's say that you define an agent with two plans to light a fire. The first plan uses the action act blowOnWood while the other one uses the action act createSpark. Now, for the sake of the argument, let's say that the first plan is triggered but the action blowOnWood fails to light the fire up, the intention will then fail, be backtracked and the other plan will be triggered and actually succeed in lightning the fire up.

Now that the base architecture is complete, we can analyze what are the API that a designer who wants to create an agent can use:

TABLE 3.1: Simple Agent API

API	DESCRIPTION	USAGE
Event && (PreCond) => [Actions].	Structure of a plan	add turn(light) && (belief light(off)) => [act gotoLight, act turnLightOn].
:- op(497,fx,desire).	Operator used to mark a term as desire	desire turn(light).
:- op(497,fx,belief).	Operator used to mark a term as belief	belief light(off).
:- op(500,fx,act).	Operator used to call a C# method (either from the agent's script or a referenced script) from Prolog	act turnLightOn act (turnLightOn,R) act (@Ref,turnLightOn) act (@Ref,turnLightOn,R)
check_plan/2	Predicate used to check the presence of a plan into the knowledge base	check_plan(add some(desire),FullPlan)
check_belief/1	Predicate used to check the presence of a belief into the knowledge base	check_belief(light(on))
add_belief/1	Predicate used to add a belief into the knowledge base	add_belief(light(on))
del_belief/1	Predicate used to remove a belief from the knowledge base	del_belief(light(off))
check_desire/1	Predicate used to check the presence of a desire into the knowledge base	check_desire(turn(light))
add_desire/1	Predicate used to add a desire into the knowledge base	add_desire(turn(light))
del_desire/1	Predicate used to remove a desire from the knowledge base	del_desire(turn(light))

Let's see an example of what is possible to realize using the above API:

```
belief name(chillyAgent).
belief have_wood.

desire world_peace.

add feeling(warm) && (belief have_wood) => [
   act searchFireplace,
   act goToFireplace,
   add_belief(atFireplace),
   act getWarm
].
```

This is the brain of the agent and will be inserted in a file with the extension ".prolog".
What this agent believe is that it has some wood and that its name is chillyAgent, it also desires for world peace but unfortunately it does not have any plan to fulfill that. It has, though, a plan to warm itself up and it only need some wood to do that. This plan will remain dormant until the body (i.e. the C# part of the agent) sense the desire of feeling warm, adding that desire to its brain.
Of course this is just a non-formal interpretation of this agent but that is exactly the way this language has been designed: clean to write, intuitive to read and close enough to the natural language to express easily complex artificial intelligence.

It is worth nothing that as soon as the plan **"add feeling(warm)"** is triggered, it will be completely executed in one single frame but actions like, for example, "gotoFireplace" could take a while to complete (i.e. you could start a coroutine inside of them) and there is no way to know if, when you add the belief "atFireplace", the agent is actually in front of the fireplace.

This simple architecture comes with some limitations:

- there is yet no support for coroutines (heavily used by Unity game designers);
- the execution of a plan is done in a single frame - i.e. if an act has an infinite loop inside, that frame will never end;
- it is possible to add or delete desires from within a plan but if there is another plan that matches that event, the current plan will be dropped in favor of the new one;
- when adding or removing a belief from the knowledge base, only the first plan that meet the requirements will be turn into an intention using the check_desires/0 predicate that means, if two or more plans are made available after the alteration of the belief set, only the first one will be executed while the others will have to wait for another recheck (i.e. another belief set alteration).

3.3.2 Unity and C#

While designing the structure of an agent using the reasoner defined before can be a clean and elegant solution, the result is just half of the process because it still

lacks the core implementation of the agent and the interaction with the Unity environment.

In the Unity framework, every entity that is placed on a scene is a **GameObject**. GameObjects are dynamic collections of Components whose number can vary at runtime. Among the types of Components that are available to the programmer, Scripts are what we need to focus our attention to. A **Script** is a C# class which extends MonoBehaviour. Scripts are used to control the GameObjects on which they reside and make them interact with the environment.
What we now need is to design a Script that can load the behaviour, created in Prolog through the reasoner (i.e. the .prolog file), and interact with it (e.g. adding/removing beliefs and desires) in order to trigger the logical reasoning that is necessary to carry on the simulation.

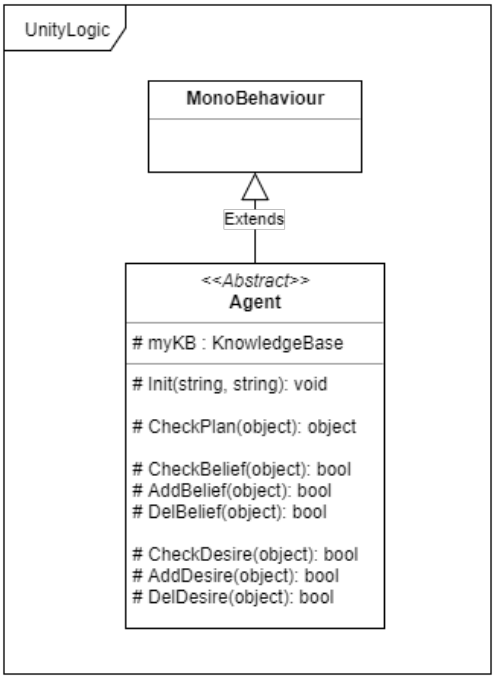

FIGURE 3.3: Architecture of a Simple Agent

The proposed architecture features an **Agent** class with a field containing the knowledge base (the brain) of the agent, a procedure to initialize it and some methods to check the existence of, add or remove a particular belief/desire. It is also possible to check the existence of a certain plan providing the head of such plan (e.g. add some(desire)).

This class shouldn't be used as a proper agent, an extension is necessary, because to initialize the knowledge base, a path and a name are required in order to load the right Prolog file but, MonoBehaviour classes don't support Constructors so it will be the programmer choice to decide when to call the Init method. Good entry points for initialization are the Awake and Start event functions.
Another purpose of the Init method is to set the reference of the Script inside the knowledge base and, since underneath an "act" operator uses this reference to

call a method that reside in the script, this step is fundamental.

```
protected void Init(string kbPath, string kbName)
{
    myKB = new KnowledgeBase(kbName, gameObject);
    myKB.Consult(kbPath);
    myKB.IsTrue(new ISOPrologReader("init.").ReadTerm(), this);
}
```

The last purpose of the Script is to contain the implementation of every "act" operation: if we have a plan with the action "act gotoFireplace" in it, a method with the name "GotoFirePlace" (case insensitive) must exist in the C# Script.

3.4 A More Advanced Agent

Agents resulting from the reasoner defined in the section 3.3 are quite expressive but there is no way to spread the execution of an intention over multiple frames and, for complex behaviour, it is a very important feature to have.
In this section I will analyze the various mechanism that are available to implement such functionality and propose an extension to the previously developed reasoner.

3.4.1 Prolog

There are at least three different approaches to realize asynchronous procedures using a game engine strongly based on an event loop-like architecture:

- execute an intention in a thread separated from the main loop;
- use an external library, like Task, to define asynchronous procedures;
- give designers a coroutine support so that they can use them within a plan.

Since Unity has a safeguard which forbids the execution of code that accesses GameObject properties outside the main loop, the first option is basically not feasible to implement and, even if someone could think of a way to synchronize the separate thread with the main one every time a critical access is made, it would render the code of the agents a lot more complicated because of the issues one usually face while trying to solve problems that require asynchronous programming (e.g. critical races).

Most of the problems of using libraries like Task to be able to use keywords like await and async to define asynchronous procedures, are correlated to the previous point. Usually these kind of libraries create a pool of threads and use them to manage the execution of the methods marked as asynchronous, which means that is still impossible to access GameObject properties from these methods.

Because of the reasons above and the widespread of coroutines in Unity, I decided to extend the reasoner with coroutine support.

The diffusion of coroutines in Unity is due to the possibility of spreading time consuming code over multiple frames using the yield keyword. Underneath,

Unity will try to execute a step of every active coroutine in the main thread and because of this, the programmer will still be able to access GameObject properties from within one.

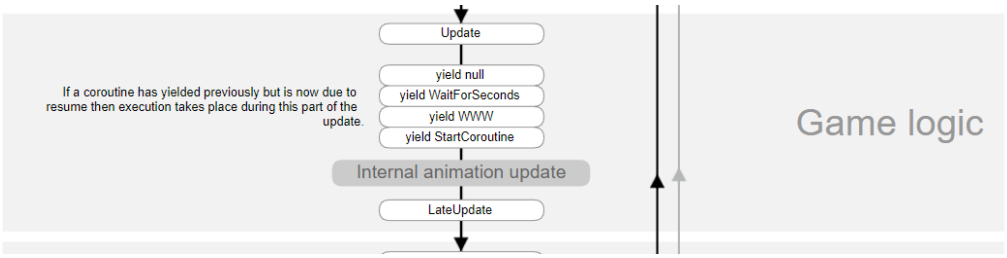

FIGURE 3.4: Coroutine Scheduling - Extracted from [Uni17]

To extend the reasoner, the first thing to do is the creation of a brand new operator to declare long C# procedures from within a plan and I decided to call it **"cr"** as in CoRoutine:

```
:- op(500,fx,cr).
```

When trying to define a behaviour for the cr operator, a problem arises: if I let Unity handle the execution of the coroutine, I will lose the declarative style that I am aiming to achieve. This happens because, to insert a coroutine inside the scheduler, you need to call the method that Unity provides, **StartCoroutine**, but this method returns as soon as the coroutine yields, whereas, to continue the intention, I need for the coroutine to completely finish its execution.
To Further explain this problem let's imagine that we have defined a plan for an agent that contains the following instructions:

```
add turn_light(on) && light(off) => [
    cr gotoLight,
    act turnLightOn
].
```

If I want Unity to handle the execution of the coroutines inside an intention, I will need to map the cr operator into the C# method StartCoroutine but, as said before, that method returns successfully as soon as it reaches the first yield instruction and as consequence, as soon as the agent starts moving toward the light (activating the coroutine named gotoLight) it will try to turn the light on (using the method named turnLightOn) even if it's still miles away from the switch.

What I need to maintain the declarative style, is to manually schedule the coroutines myself, pausing the intention until the coroutine has completed its execution. In order to do so, I need the reasoner to store two things every time it reaches a cr operator: the reference to the coroutine and the continuation of the intention (i.e. every instruction after the cr).

An important note: the reason why I need to design a manual scheduler is because Prolog runs on the main thread and if I suspend the intention (e.g. suspending the thread) I will also block the current frame and the entire game.

A very handy way to store frequently accessed information is to use the Exclusion Logic so, when the reasoner stumbles upon a cr operator it can simply assert the coroutine reference into the knowledge base and do the same for the continuation of the intention.
Using the Exclusion Logic it is possible to store values in a tree-like structure and easily update a node value simply asserting the new one at the same path.

It is now possible to define a Prolog fact to use in order to save the data we need to execute a plan:

```
set_active_task(A) :-
    (A = (@Ref,Name,Cont), !,
        del_active_task,
        get_coroutine_task((@Ref,Name,Task)),
        assert(/active_task/task:Task),
        assert(/active_task/cont:Cont))
    ;
    (A = (Name,Cont), !,
        del_active_task,
        get_coroutine_task((Name,Task)),
        assert(/active_task/task:Task),
        assert(/active_task/cont:Cont))
    ;
    (A = (Cont), !,
        del_active_task,
        assert(/active_task/task:null),
        assert(/active_task/cont:Cont)).
```

It is worth notice that the predicate **set_active_task/1** has a single parameter but inside will try to match it with some patterns to understand how many information were provided. This is just a stylistic choice to have a cleaner code and only one predicate instead of three.

The first match of this predicate will be used in case we want to provide the location of the coroutine, manually - i.e. the agent does not have that coroutine inside of its script but knows where is possible to find it, while the second match will search inside the agent script. Name and Cont are respectively the name of the coroutine and the rest of the intention.

The third match possible for this predicate requires only a single parameter that is the rest of the intention to execute, this is useful when the reasoner stumbles upon any action that is not a cr operation. This option has been implemented after an important design decision: how much of an intention should be executed in a single frame.
There are some advantages in having the intention executed all in one frame, for example in case of failure it will be possible to have complete backtracking of the actions already completed but, with the advent of coroutines, this is not feasible anymore since they are executed one step at a frame and every frame is a new Prolog interrogation because UnityProlog cannot be paused in between frames since

it runs on the same thread of the simulation: if UnityProlog thread is paused, all the game is paused.
The alternative is to render every action atomic: one action executed every frame. This is a more feasible and correct solution but we will loose the possibility of backtracking a plan in case of failure and this is why, later on, this feature will be manually implemented.

In the section 3.3, there was some sort of partial transactionality: when an intention failed, everything but "act" actions (i.e. C# methods) could be rollbacked. Now, that partial transactionality is lost but, since realize a full transactionality is not possible (some environment alterations aren't reversible) I decided to leave this feature to future analysis to understand if it could bring some semantic enhancement to the language and if it's actually feasible to implement.

```
get_coroutine_task(A) :-
    (A = (@Ref,Name,Task), !,
        call_method(Ref,Name,Task))
    ;
    (A = (Name,Task), !,
        get_ref(Ref),
        call_method(Ref,Name,Task)).
```

The predicate **get_coroutine_task/1** is used to get the enumerable that is the coroutine with the same name as the Name parameter. It works in the same way of set_active_task/1: only one parameter but multiple matches possible. If we provide three parameters, the first one will be the reference of the script where to find the coroutine named Name.

```
del_active_task :-
    assert(/active_task/task:null),
    assert(/active_task/cont:null).
```

Lastly, the predicate **del_active_task/0** is used as a safeguard to clean previously allocated task and continuation.

An extension to the predicate extract_task/1 is also necessary to make the reasoner able to handle coroutines:

```
extract_task(A) :-
    (A = [cr (@Ref,M)|N], !,
        set_active_task((@Ref,M,N)))
    ;
    (A = [cr M|N], !,
        set_active_task((M,N)))
    ;
    (A = [M|N], !,
        M,
        set_active_task((N)))
    ;
    del_active_task.
```

With this extension the reasoner is now able to extract a single action from the intention, execute it and save the continuation to carry on the intention in the next frame. In case it fails to match any pattern (i.e. intention completed or failed), it will simply clean the task and continuation of the current intention. To understand what is the purpose of this, we need to introduce a new predicate which will be used to continue the reasoning cycle of an agent: if there is an active intention not completed or some pending alternative plan, go on with that, if not, try to fulfill your desires.

```
go_on :-
    /active_task/task:Task,
    Task \= null,
    call_method(Task,'MoveNext'(),Ret),
    (Ret = false, assert(/active_task/task:null) ; true).

go_on :-
    /active_task/cont:Cont,
    Cont \= null,
    task_completed.

go_on :-
    /active_task/plans:Plans,
    Plans = [H|T],
    assert(/active_task/plans:T),
    set_active_task(H).

go_on :-
    desire D,
    add D && _ => _,
    findall(
        X,
        (
            add D && C => A,
            append([C],A,Temp),
            append(Temp,[del_desire(D)],X)
        ),
        Plans
    ),
    assert(/active_task/plans:Plans).
```

The reasoning cycle has been broken down in four stages and will be called one time at a frame:

- check if there is an active task (i.e. a coroutine) and execute a step of it;
- if there isn't an active task, check if there is some continuation to carry on and execute a step of it;
- if no task and no continuation are set, check if there is some alternative plan to execute;
- if none of the above conditions are verified, check if there is a desire to fulfill and save all plans that matches that desire.

task_completed/0 will be called if there is no task pending but the intention has still some actions left to execute:

```
task_completed :-
    /active_task/cont:Cont,
    del_active_task,
    extract_task(Cont).
```

Since now is the go_on/0 predicate that brings on the reasoning cycle, we can further simplify the previously defined predicates for the alteration of the belief and desire set:

```
add_belief(B) :-
    assert(belief B).

del_belief(B) :-
    retract(belief B).

add_desire(D) :-
    desire D
    ;
    assert(desire D).

del_desire(D) :-
    retract(desire D),
    (
        del D && C => A, !,
        C,
        set_active_task(A)
        ;
        true
    ).
```

When you add a desire the go_on/0 predicate will check whenever it can, if a plan can fulfill it but the deletion is not that trivial: you can't check for something that it isn't there. This implementation of the del_desire/1 predicate tries to work around this problem but it's still not an optimal solution because if there is a plan that matches the deletion event, the current intention will be dropped in favor of the new one. A further extension of the predicate will be left as future work.

One last thing that I found to be very useful to have in this new reasoner, is the possibility to stop the execution of alternative plans. When an intention either fails or completes, the go_on/0 predicate will try to execute, if there is one, the next alternative plan (i.e. a plan with the same event that the active intention has) but there are some cases when this is not the desired behaviour and we want stop any other alternative:

```
stop :-
    assert(/active_task/plans:null).
```

The **stop/0** predicate can be used as an action inside of a plan to delete all pending alternatives after that plan fails or completes.

Here are listed all the API available for the designer:

TABLE 3.2: Advanced Agent API

API	DESCRIPTION	USAGE
Event && (PreCond) => [Actions].	Structure of a plan	add turn(light) && (belief light(off)) => [cr gotoLight, act turnLightOn, stop].
:- op(497,fx,desire).	Operator used to mark a term as desire	desire turn(light).
:- op(497,fx,belief).	Operator used to mark a term as belief	belief light(off).
:- op(500,fx,act).	Operator used to call a C# method (either from the agent's script or a referenced script) from Prolog	act turnLightOn act (turnLightOn,R) act (@Ref,turnLightOn) act (@Ref,turnLightOn,R)
:- op(500,fx,cr).	Operator used to call a C# coroutine (either from the agent's script or a referenced script) from Prolog	cr gotoLight cr (gotoLight,R) cr (@Ref,gotoLight) cr (@Ref,gotoLight,R)
stop/0	Predicate used to expressively stop backtracking of an intention	stop
check_plan/2	Predicate used to check the presence of a plan into the knowledge base	check_plan(add some(desire),FullPlan)
check_belief/1	Predicate used to check the presence of a belief into the knowledge base	check_belief(light(on))
add_belief/1	Predicate used to add a belief into the knowledge base	add_belief(light(on))
del_belief/1	Predicate used to remove a belief from the knowledge base	del_belief(light(off))
check_desire/1	Predicate used to check the presence of a desire into the knowledge base	check_desire(turn(light))
add_desire/1	Predicate used to add a desire into the knowledge base	add_desire(turn(light))
del_desire/1	Predicate used to remove a desire from the knowledge base	del_desire(turn(light))

3.4.2 Unity and C#

Regarding the C# side of the agent, there is only one, but significant, alteration to the architecture defined in sec. 3.3.
We have now a reasoning cycle that handles the execution of an intention one action at a time, which means that this cycle must be triggered manually to completely fulfill a desire.

In this case I will mimic what Unity already does while handling coroutines, that is execute a step once per frame, more specifically, after the Update function has terminated but before the LateUpdate function is called. The easiest way to implement this is to call the Prolog entry point go_on/0 as the first instruction of the LateUpdate:

```
public void LateUpdate()
{
    myKB.IsTrue(new ISOPrologReader("go_on.").ReadTerm(), this);
}
```

The only thing left to do is to update the architecture, adding the LateUpdate function to the Agent:

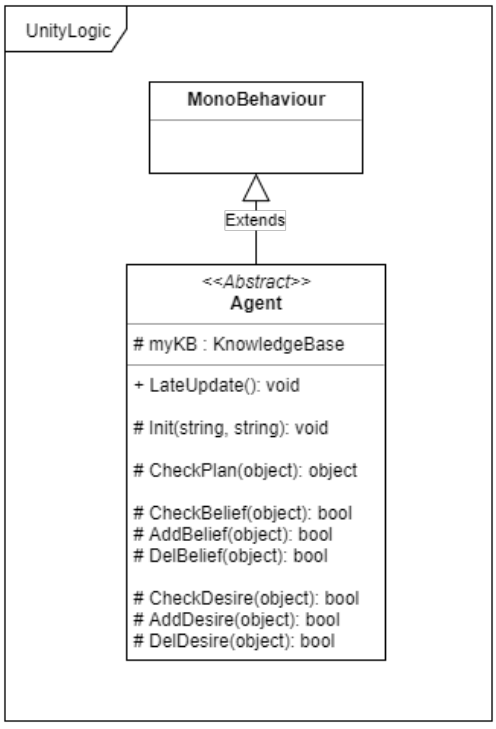

FIGURE 3.5: Architecture of an Advanced Agent

3.5 A Simple Artifact

In games and MAS simulations the environment plays a fundamental role. Agents often need to interact with passive objects placed all around the scene, for example a character in a house that needs to use the fridge. A way to represent the

fridge is needed. Obviously these environmental objects could be modeled as agents that do nothing except when real agents interact with them but that would be an abuse of semantic because objects and agents are fundamentally different. That is why I decided to introduce the Artifact concept in this system, inspired by the A&A meta-model [ORV08] and the CArtAgO framework [RVO07].

3.5.1 Prolog

An artifact should provide to its users plans and knowledge. Let's take a light switch as an example: an agent that wants to use the switch should be informed with a list of plans that it provides, turn on and turn off, and its current state, on or off.

Partially reusing the reasoner of the agents, I designed the plans of an artifact as follows:

```
plan Name && (PreConditions) => [ListOfActions]
```

The only difference with an agent's plan is the lack of an event that could trigger it and this is mainly because of the passive nature of an artifact: it can be used by an agent but it cannot reason on its own.
The **"plan"** keyword is simply an operator used to distinguish knowledge from plans:

```
:- op(497,fx,plan).
```

Name is the name of the plan (e.g. light(fire)). **PreConditions** and **ListOfActions** have the exact same structure of an agent's plan but PreConditions will be verified by the agent who interacts with the artifact, while **ListOfActions** changes its meaning based on what type of interaction the agent decide to have with the artifact. An agent could ask the artifact to activate a plan, in this case, ListOfActions will be executed by the artifact using its own C# script or could ask the artifact to use a plan, this means that the agent will be the executor of ListOfActions.

```
activate(Name,Conditions) :-
    plan Name && C => A,
    (
        set_active_task(A),
        Conditions = C
        ;
        Conditions = false
    ).

use(Name,Plan) :-
    plan Name && C => A,
    append([C],A,Plan).
```

With the "activate" semantic it is possible to create reactive artifacts: passive entities waiting to be triggered by agents.

Regarding the representation of knowledge for the artifact, the simple reusing of the concept of belief will do. There is no actual need of creating a new concept because it would have the exact same semantic of an agent's belief.

An artifact brain can look very similarly to an agent brain:

```
belief type(fireplace).

plan light(fire) && (belief have_wood) => [
    act placewood,
    del_belief(have_wood),
    cr lightfire
].
```

An agent that wants to use this plan must know in its own script the method named "placewood" and the coroutine "lightfire" and must wait for the plan to be fully executed before it can do anything else, if it decides instead to activate the plan, the artifact becomes effectively independent until the plan is completed (every action of the plan resides in the artifact script) which means that an agent that activates a plan does not have to wait for the plan to be completed.

Since an artifact can become an active entity, it is necessary to provide an entry point to carry on an active plan similarly to what has been done with agents:

```
go_on :-
    /active_task/task:Task,
    Task \= null,
    call_method(Task,'MoveNext'(),Ret),
    (Ret = false, assert(/active_task/task:null) ; true).

go_on :-
    /active_task/cont:Cont,
    Cont \= null,
    task_completed.
```

Differently from an agent, an artifact would not try to fulfill its desires because it has none, so that phase of the reasoning cycle has been removed.

The API made available for the designing of an Artifact are the follows:

TABLE 3.3: Simple Artifact API

API	DESCRIPTION	USAGE
:- op(497,fx,plan).	Operator used to define plans	plan turn(light) && (belief light(off)) => [cr gotoLight, act turnLightOn, stop].
:- op(497,fx,belief).	Operator used to mark a term as belief	belief light(off).
:- op(500,fx,act).	Operator used to call a C# method (either from the agent's script or a referenced script) from Prolog	act turnLightOn act (turnLightOn,R) act (@Ref,turnLightOn) act (@Ref,turnLightOn,R)
:- op(500,fx,cr).	Operator used to call a C# coroutine (either from the agent's script or a referenced script) from Prolog	cr gotoLight cr (gotoLight,R) cr (@Ref,gotoLight) cr (@Ref,gotoLight,R)
stop/0	Predicate used to expressively stop backtracking of an intention. It has semantic only when a plan is used and not when activated	stop
check_belief/1	Predicate used to check the presence of a belief into the knowledge base	check_belief(light(on))
add_belief/1	Predicate used to add a belief into the knowledge base	add_belief(light(on))
del_belief/1	Predicate used to remove a belief from the knowledge base	del_belief(light(off))

3.5.2 Unity and C#

Similar to the agent counterpart, an Artifact is an abstract class that should be extended before creating a new type of artifact for the same initialization problem already explained in the section 3.3.

This artifact makes use of the same LateUpdate event function that the Agent uses as an entry point to carry on the reasoning cycle and this is because of the

possible reactive nature of an artifact when it is activated instead of simply used. The Init function is also the exact same of the Agent and it is used to create and load the knowledge base of the Artifact.

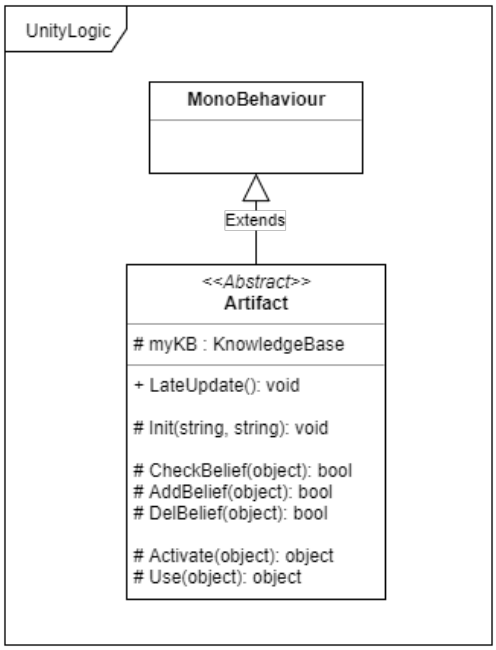

FIGURE 3.6: Architecture of a Simple Artifact

This simple architecture has been designed with in mind the interface that an artifact should expose to the agents. AddBelief and DelBelief methods will be used by an agent that wants to alter the artifact belief set. Activate and Use methods are used to change the nature of the artifact behaviour: as said before, if an agent activates an artifact, the latter will become active to carry on the requested plan on its own, on the other hand, if the agent simply uses the artifact, this would not affects its passive nature and it is the agent itself to carry on the plan that it has requested.

3.6 Communication Layer

In a Multi-Agent System, communication plays a fundamental role: agents must have ways to interact with each other and with the environment. This section will cover Agent-to-Agent and Agent-to-Artifact communication.

An agent should be able to check and modify an artifact belief set and to use or activate one of the plans it provides.

```
learn_artifact_belief(Ref,B) :-
    ref_to_artifact(Ref,Art),
    call_method(Art,'CheckBelief'(B),Ret),
    Ret \= false,
    add_belief(B).

check_artifact_belief(Ref,B) :-
    ref_to_artifact(Ref,Art),
    call_method(Art,'CheckBelief'(B),Ret),
    Ret \= false.

add_artifact_belief(Ref,B) :-
    ref_to_artifact(Ref,Art),
    call_method(Art,'AddBelief'(B),Ret),
    Ret \= false.

del_artifact_belief(Ref,B) :-
    ref_to_artifact(Ref,Art),
    call_method(Art,'DelBelief'(B),Ret),
    Ret \= false.

activate_artifact(Ref,PlanName) :-
    ref_to_artifact(Ref,Art),
    call_method(Art,'Activate'(PlanName),Ret),
    Ret.

use_artifact(Ref,PlanName,Ret) :-
    ref_to_artifact(Ref,Art),
    call_method(Art,'Use'(PlanName),Ret),
    Ret \= false.
```

An agent is able to check if a belief is present in an artifact knowledge base through **check_artifact_belief/2** and also learn a belief (i.e. add that belief into its own knowledge base) through the **learn_artifact_belief/2** predicate. An agent is also able to add or delete beliefs from an artifact (**add_artifact_belief/2** and **del_artifact_belief/2**) and use or activate a plan knowing its name (i.e. providing the correct PlanName parameter) using respectively **use_artifact/3** or **activate_artifact/2**.

The predicate use_artifact/3 has three parameters because the last one is used to contains the preconditions of a plan that the agent itself should verify (see sec. 3.5) but to avoid the programmer the burden of check those preconditions manually, I decided to modify the extract_task/1 predicate to handle the checking automatically, leaving the programmer a more simple predicate use_artifact/2 to use:

```
extract_task(A) :-
    (A = [use_artifact(Ref,Action)|N], !,
        use_artifact(Ref,Action,Ret),
        append(Ret,N,Res),
        set_active_task(Res))
    ;
    (A = [cr (@Ref,M)|N], !,
        set_active_task((@Ref,M,N)))
    ;
    (A = [cr M|N], !,
        set_active_task((M,N)))
    ;
    (A = [M|N], !,
        M,
        set_active_task((N)))
    ;
    del_active_task.
```

Using this approach, a programmer that wants to use an artifact plan from an agent, should simply use the predicate use_artifact/2 that will be interpreted by the reasoner and mapped into use_artifact/3.

Regarding the Agent-to-Agent interaction, an agent should be able to check, learn and possibly alter the belief set of another agent:

```
learn_agent_belief(Ref,B) :-
    ref_to_agent(Ref,Ag),
    call_method(Ag,'CheckBelief'(B),Ret),
    Ret \= false,
    add_belief(B).

check_agent_belief(Ref,B) :-
    ref_to_agent(Ref,Ag),
    call_method(Ag,'CheckBelief'(B),Ret),
    Ret \= false.

add_agent_belief(Ref,B) :-
    ref_to_agent(Ref,Ag),
    call_method(Ag,'AddBelief'(B),Ret),
    Ret \= false.

del_agent_belief(Ref,B) :-
    ref_to_agent(Ref,Ag),
    call_method(Ag,'DelBelief'(B),Ret),
    Ret \= false.
```

The same goes for the desire set:

```
learn_agent_desire(Ref,D) :-
    ref_to_agent(Ref,Ag),
    call_method(Ag,'CheckDesire'(D),Ret),
    Ret \= false,
    add_desire(D).

check_agent_desire(Ref,D) :-
    ref_to_agent(Ref,Ag),
    call_method(Ag,'CheckDesire'(D),Ret),
    Ret \= false.

add_agent_desire(Ref,D) :-
    ref_to_agent(Ref,Ag),
    call_method(Ag,'AddDesire'(D),Ret),
    Ret \= false.

del_agent_desire(Ref,D) :-
    ref_to_agent(Ref,Ag),
    call_method(Ag,'DelDesire'(D),Ret),
    Ret \= false.
```

Moreover, an agent should also be able to learn plans from others:

```
learn_agent_plan(Ref,Head,Raw) :-
    ref_to_agent(Ref,Ag),
    (Head = add _ ; Head = del _),
    call_method(Ag,'CheckPlan'(Head),Ret),
    Ret = Head && C => A,
    (
        Raw, !, assert(Ret)
        ;
        convert_plan(Ag,A,Converted),
        assert(Head && C => Converted)
    ).
```

To learn a plan it is necessary to provide the **Head** of that plan (i.e. Event section of a plan, "add some(desire)" for example), the reference of the other agent (i.e. **Ref** parameter) and specify the way this plan should be learned. An agent can learn a plan as is (providing the **Raw** parameter as true) which means that should know every single action that the requested plan contains or with the reference of the other agent (Raw parameter as false) hard-copied in every "act" and "cr" action, which means that those actions will be handled by the other agent script as procedure calls.
Let's analyze a simple example to further explain this feature:

1. Agent A asks Agent B to learn its plan to turn a light on;
2. Agent A knows that the actions required to turn a light on are not in its Script so it will provide the Raw parameter as false;

3. Now Agent A knows how to turn a light on but every time it needs to execute an "act" or a "cr" action contained in the learned plan, it will procedurally call methods that are present in the Agent B's Script.

This feature could be destructive if not used carefully: when Agent A procedurally calls a method that resides within the Agent B's Scripts, it could potentially change the internal state of the agent, if that's the purpose of such method. It is advisable to learn plans by reference only if all the actions don't alter the internal state of the agent.

We can now take a look at the full list of API for enabling the communication among agents and artifact that a programmer can use:

TABLE 3.4: Agent-To-Artifact Communication API

AGENT-TO-ARTIFACT		
API	**DESCRIPTION**	**USAGE**
learn_artifact_belief/2.	Used to learn a belief from an Artifact, knowing its reference	learn_artifact_belief(Ref, some(belief))
check_artifact_belief/2.	Used to check if an Artifact have a particular belief, knowing its reference	check_artifact_belief(Ref, some(belief))
add_artifact_belief/2.	Used to add a belief into an Artifact knowledge base, knowing its reference	add_artifact_belief(Ref, some(belief))
del_artifact_belief/2.	Used to remove a belief from an Artifact knowledge base, knowing its reference	del_artifact_belief(Ref, some(belief))
activate_artifact/2.	Used to activate an Artifact plan whose reference is known, rendering it independent until the execution of the requested plan is over	activate_artifact(Ref, plan(name))
use_artifact/2.	Action that an agent who knows an artifact reference can do to be able to use a particular plan of such artifact	use_artifact(Ref, plan(name))

TABLE 3.5: Agent-To-Agent Communication API

AGENT-TO-AGENT		
API	**DESCRIPTION**	**USAGE**
learn_agent_belief/2.	Used to learn a belief from an Agent, knowing its reference	learn_agent_belief(Ref, some(belief))
check_agent_belief/2.	Used to check if an Agent have a particular belief, knowing its reference	check_agent_belief(Ref, some(belief))
add_agent_belief/2.	Used to add a belief into an Agent knowledge base, knowing its reference	add_agent_belief(Ref, some(belief))
del_agent_belief/2.	Used to remove a belief from an Agent knowledge base, knowing its reference	del_agent_belief(Ref, some(belief))
learn_agent_desire/2.	Used to learn a desire from an Agent, knowing its reference	learn_agent_desire(Ref, some(desire))
check_agent_desire/2.	Used to check if an Agent have a particular desire, knowing its reference	check_agent_desire(Ref, some(desire))
add_agent_desire/2.	Used to add a desire into an Agent knowledge base, knowing its reference	add_agent_desire(Ref, some(desire))
del_agent_desire/2.	Used to remove a desire from an Agent knowledge base, knowing its reference	del_agent_desire(Ref, some(desire))
learn_agent_plan/3.	Used to learn a plan from an Agent knowledge base, knowing its reference	learn_agent_plan(Ref, add some(desire), true) learn_agent_plan(Ref, add some(desire), false)

Chapter 4

Recycling Robots: A Case Study

We have, at this point, every tool necessary to define a simulation that comprises most of the crucial aspects of a Multi-Agent System that also exploits artifacts as environmental objects.
As a final analysis, I will try to replicate the same behaviours resulting from this new defined language, using the classic approach that most Unity programmers take advantage of to create AI: Finite State Machines (FSM).

4.1 Requirements

Closed environment filled with objects that need to be recycled: paper, plastic and glass.
Every object is an Artifact with different information in it:

- Paper: contains a belief that expresses the type paper;
- Plastic: contains a belief that expresses the type plastic;
- Glass: contains a belief that expresses the type glass, moreover, it has a plan with recycling instructions.

Three robots in the closed environment, each one of them is an Agent with different capabilities:

- Learning Robot: doesn't know how to recycle anything but is able to analyse objects and ask other robots when in doubt, it can learn plans and delegate actions;
- Plastic Robot: can recycle objects marked as plastic but nothing more, if it encounters an object different from plastic, it will ignore it;
- Paper Robot: can recycle objects marked as paper but nothing more, differently from the other robots, it does nothing unless someone else asks for its help.

To recycle an object a robot should simply bring it to the correct bin.

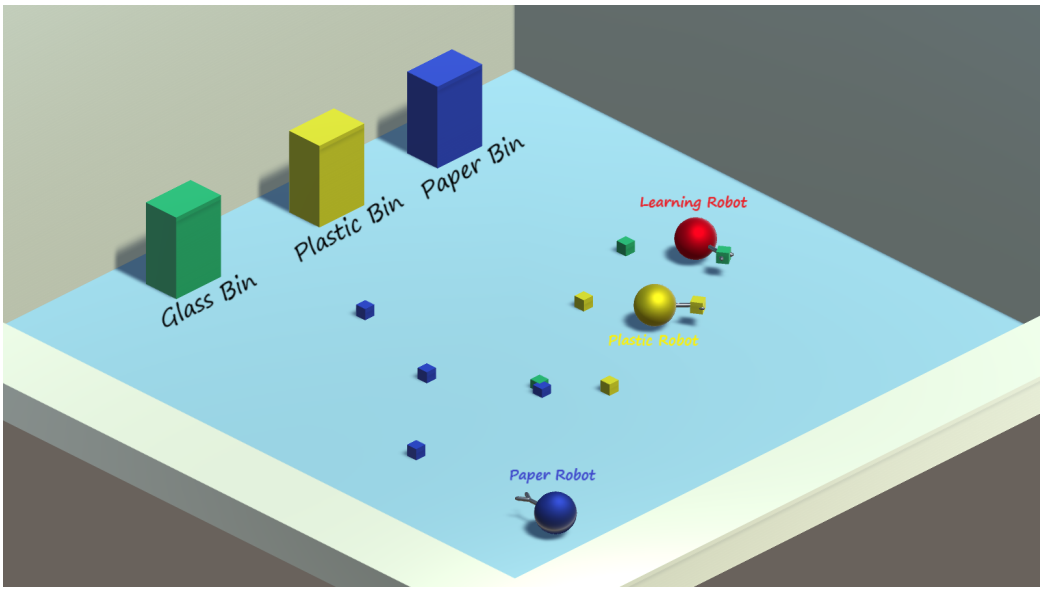

FIGURE 4.1: Recycling Robots Unity Scene

4.2 Experimental Setup

As said before, to attain meaningful results, I need to divide the experimental setup in two phases. First I will develop this case study using entirely the BDI reasoner previously designed, then I will remove the reasoning cycle from the agents and artifacts and replace it with FSMs. What I expect is a more clean and intuitive design resulting from the exploitation of the reasoning cycle in opposition to the same design but implemented with Finite State Machines.

4.2.1 BDI Agents and Artifacts

Let's start defining the more complex Artifact in the environment: Glass.
The glass artifact has a belief and a plan that explains to an agent how to recycle it:

```
% GLASS ARTIFACT

belief type(glass).

plan recycle && true => [
    act (searchBin("Glass"),Bin),
    Bin \= false,
    cr goto(Bin),
    act recycle,
    del_belief(hand(_))
].
```

What this plan says is:

1. search the reference for the Glass bin
2. if you did found the bin then...
3. go to (move toward) the bin

4. recycle the object
5. empty your hand

The other two artifacts don't have plans inside of them and the only thing that changes is the belief named type:

```
% PAPER ARTIFACT

belief type(paper).
```

```
%PLASTIC ARTIFACT

belief type(plastic).
```

We can now take a look at the agents starting with the less complex of the three, the Paper Robot:

```
add recycle(G) && (\+ belief hand(_)) => [
    (
        not(check_artifact_belief(G,busy)),
        add_artifact_belief(G,busy)
    ),
    check_artifact_belief(G,type(Type)),
    Type = paper,
    cr goto(G),
    act pickup(G),
    add_belief(hand(G)),
    act (searchBin("Paper"),Bin),
    Bin \= false,
    cr goto(Bin),
    act recycle,
    del_belief(hand(_))
].
```

The Paper Robot has no desire, which means it will do nothing until one is added to its knowledge base.
What its recycling plan says is:

1. when it has the desire to recycle something (i.e. G, as the artifact reference) and its hand is empty then...
2. check in a single action (two or more actions embraced by brackets means that they are executed in the same frame) if that artifact G is available and if it is, mark it as busy
3. extract the type from the artifact
4. check if it's paper
5. move toward the artifact G

6. pick it up
7. remember that the hand is full from now on, in case of possible failures
8. search for the Paper bin
9. if that bin exists then...
10. move toward the bin
11. recycle the artifact
12. clear the hand

The Plastic Robot is very similar to the Paper Robot but it has an initial desire to work and also a couple of plans to handle objects that it does not know how to recycle:

```
desire work.

add work && true =>[
    add_desire(recycle)
].
add recycle && (\+ belief hand(_)) => [
    act (searchGarbage,G),
    G \= false,
    (
        not(check_artifact_belief(G,busy)),
        add_artifact_belief(G,busy)
    ),
    cr goto(G),
    act pickup(G),
    add_belief(hand(G)),
    check_artifact_belief(G,type(Type)),
    Type = plastic,
    add_desire(recycle(plastic)),
    stop
].
add recycle(plastic) && (belief hand(_)) => [
    act (searchBin("Plastic"),Bin),
    Bin \= false,
    cr goto(Bin),
    act recycle,
    del_belief(hand(_)),
    add_desire(work),
    stop
].
add recycle && (belief hand(G)) => [
    act dropdown,
    del_belief(hand(_)),
    del_artifact_belief(G,busy),
    add_desire(work),
    stop
].
```

A general explanation on how those plans work should suffice to give an idea of the robot behaviour:

1. the purpose of the first plan is simply to trigger the main behaviour of the robot - i.e. recycling
2. the second plan handles the search of an artifact - if it finds a plastic artifact it will trigger the third plan
3. the third plan is called when the hand of the robot is full and contains a plastic artifact, its purpose is to search a plastic bin and to recycle the artifact in its hand
4. the fourth and last plan is finally called if all the previous plans have failed at some point - the robot will drop the artifact in its hand and go back to work

It is worth notice that every plan but the first one terminates with the stop/0 predicate which means that, if any of those successfully reaches the end, all the pending alternatives will be discarded. This has been done because the plans are designed as alternatives and not as continuations of some sort: if the robot successfully completes the second plan it means that it was able to find an artifact that can recycle but the fourth plan is still pending (they have the same Event), waiting to be executed, and since its purpose is to drop the current artifact and start to work from the start, I do not want this to be triggered.

Regarding the Learning Robot, its behaviour is an extension of the Plastic Robot:

1. if the Learning Robot finds an artifact that does not know how to recycle it will try to check if there is a plan inside the artifact itself with instructions
2. if no plan is found it will try to learn a plan from the Plastic Robot
3. if the Plastic Robot fails to provide an appropriate plan, the Learning Robot will drop the artifact and add a desire to the Paper Robot triggering its behaviour

```
desire work.

add work && true =>[
    add_desire(recycle)
].
add recycle && (\+ belief hand(_)) => [
    act (searchGarbage,G),
    G \= false,
    (
        not(check_artifact_belief(G,busy)),
        add_artifact_belief(G,busy)
    ),
    cr goto(G),
    act pickup(G),
    add_belief(hand(G)),
    check_artifact_belief(G,type(Type)),
    check_plan(add recycle(Type),_),
    add_desire(recycle(Type)),
    stop
].
add recycle && (belief hand(G)) => [
    use_artifact(G,recycle),
    add_desire(work),
    stop
].
add recycle && (belief hand(G)) => [
    check_artifact_belief(G,type(Type)),
    act (findPlasticBot,Bot),
    act stopbot(Bot),
    cr goto(Bot),
    act resumebot(Bot),
    learn_agent_plan(Bot,add recycle(Type),true),
    add_desire(recycle(Type)),
    stop
].
add recycle && (belief hand(G)) => [
    act (findPaperBot,Bot),
    add_agent_desire(Bot,recycle(G))
].
add recycle && (belief hand(G)) => [
    act dropdown,
    del_belief(hand(_)),
    del_artifact_belief(G,busy),
    add_desire(work),
    stop
].
```

I will not go in detail regarding the C# Script of every agent and artifact since it is irrelevant how “cr” and “act” are implemented for the purpose of the comparison between behaviours implemented with this reasoner and those implemented making use of Finite State Machines.

4.2.2 Finite State Machines

As proof of concept, I will only express the behaviour of the Learning Robot with a FSM using the same methods and coroutines that the BDI version of it, already used.

```
void Update()
{
    switch (current)
    {
        /* [...] */

        case State.USE_OBJECT:
            var hand = GetComponentInChildren<Garbage>();
            /*
             * bool can = (bool)hand.Use(hand.type);
             * if (!can)
             * {
             *     current = State.LEARN_FROM_PLASTIC_ROBOT;
             *     return;
             * }
             * current = State.WORK;
             */
            break;

        case State.LEARN_FROM_PLASTIC_ROBOT:
            var bot = FindPlasticBot();
            StopBot((GameObject)bot);
            Goto((GameObject)bot);
            ResumeBot((GameObject)bot);
            /*
             * bool have_plan = LearnPlan(bot,type,true);
             * if(!have_plan){
             *     current = State.ACTIVATE_PAPER_BOT;
             *     return;
             * }
             * UseLearnedPlan(type);
             * Recycle(hand);
             * current = State.WORK;
             */
            break;


        /* [...] */
    }
}
```

Most of the plans of the BDI agent can easily be mapped in a state of a FSM but two of them are particularly difficult to design without the reasoner. A more thorough analysis is available in the section 4.3.

4.3 Results

While a BDI agent can use plans that artifacts provide, meaning that can read a sequence of actions and execute it, there is no easy way to express such behaviour using C# exclusively.
The easiest way to realize a BDI plan in C# is to map it into a method where every action is a hard-coded instruction but that method will be strongly coupled with the Script in which it resides: if we have a method to recycle glass inside the glass object, an agent that wants to use it, will necessarily mess with the object internal state and those actions will not affect the agent itself - i.e. a goto action inside such method will make the object move instead of the agent. There are of course workarounds to this problem, for instance the agent could pass its reference to the object that owns the method and let the object modify the agent internal state but it would make the code a lot more complex and less maintainable but, more importantly, using this strategy will result in a distortion of the "use" semantic: it will be the object that uses the agent and not the other way around.

If there is no easy way to read a plan from an object and use its actions, trying to learn a plan from an agent would be even more complicated. In this scenario, a plan is a state in the FSM of the agent, learning a new plan would mean add a new state at runtime and also modify the other states to make the new one accessible at some point and this is effectively impossible. There are ways to write and compile code at runtime, using for example the CSharpCodeProvider class but I can't assure a fully Unity compatibility and sure enough would not make programming an agent any easier.

Chapter 5

Conclusion And Future Works

The definition of declarative behaviours for BDI Agents and Artifacts using the language developed in this project shown very promising results. It is clean and intuitive to write high-level behaviours, and the reasoner has capabilities way beyond what is possible to realize using C# alone: plans of actions can be shared among agents and artifacts; agents can build their own behaviour at runtime and through the exploitation of the logic backtracking it is easy to switch from plan to plan while looking for a correct behaviour that can be used to fulfill a desire.

There are still some limitations that future works should tackle. As yet, the reasoner is only able to handle one plan at a time, meaning that agents cannot try to fulfill a new desire if there is an active intention still waiting to be completed. The language should be extended with operators that can be used to mark plans as parallel while the reasoner should be able to bring on the execution of those special plans at the same time.

The language used to define agents and artifact has been designed specifically to withstand the limitations that UnityProlog had and since most of the operators were impossible to redefine, the language itself had become more complicated that I originally planned it to be. tuProlog does not have the same limitations that UnityProlog has and when the bug that currently make it unusable will be fixed, it has the potential to bring some powerful enhancements to the language and the reasoner itself. Even though tuProlog could be promising, unlike UnityProlog, it has no direct way to interact with Unity and it also lacks the support for Exclusion Logic that was massively used in this project, which means that if the interpreter is replaced, both the language and the reasoner would need to be almost completely redesigned.

Lastly, there is an aspect that shouldn't be forgotten: performance. This approach is very powerful and can facilitate the design of autonomous agents in simulations but a thorough performance analysis is necessary when we want to use it for commercial games where every millisecond matters. The Prolog version used in this project is interpreted rather than compiled and that alone could consume quite a few runtime resources moreover, the reasoning cycle of agents and artifacts is triggered once per frame and there could be margins for optimizations.

Bibliography

[BCG07] Fabio Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent with JADE Systems*. 2007, p. 286.

[BHW07] Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. 2007, pp. 1–273.

[Dav+06] Euan M. Davidson et al. "Applying multi-agent system technology in practice: Automated management and analysis of SCADA and digital fault recorder data". In: *IEEE Transactions on Power Systems* 21.2 (2006), pp. 559–567.

[DH04] A.L. Dimeas and N.D. Hatziargyriou. "A multiagent system for microgrids". In: *IEEE Power Engineering Society General Meeting, 2004.* 2 (2004), pp. 55–58.

[DOR01] Enrico Denti, Andrea Omicini, and Alessandro Ricci. "tuProlog: A Light-weight Prolog for Internet Applications and Infrastructures". In: *Practical Aspects of Declarative Languages. 3rd International Symposium (PADL 2001), Las Vegas, Nevada, March 11–12, 2001 Proceedings*. Springer. 2001, pp. 184–198.

[ES18] Richard Prideaux Evans and Emily Short. "The AI Architecture of Versu". In: (2018).

[Eva10] Richard Evans. "Introducing exclusion logic as a deontic logic". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6181 LNAI (2010), pp. 179–195.

[Hor14] Ian Horswill. "Architectural issues for compositional dialog in games". In: *AAAI Workshop - Technical Report* WS-14-17 (2014), pp. 15–17.

[Hor17] Ian Horswill. *UnityProlog*. 2017. URL: https://github.com/ianhorswill/UnityProlog.

[KGR96] David Kinny, Michael Georgeff, and Anand Rao. "A Methodology and Modelling Technique for Systema of BDI Agents". In: *Agents breaking away* (1996), pp. 56–71.

[ORV08] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. "Artifacts in the A&A meta-model for multi-agent systems". In: *Autonomous Agents and Multi-Agent Systems* 17.3 (2008), pp. 432–456.

[Pra+03] Isabel Praça et al. "MASCEM : A Multiagent Markets". In: (2003).

[Rao96] Anand S. Rao. "AgentSpeak(L): BDI agents speak out in a logical computable language". In: L (1996), pp. 42–55.

[RG95] Anand S. Rao and Michael P. Georgeff. "BDI Agents: From Theory to Practice". In: *Proceedings of the First International Conference on Multiagent Systems* 95 (1995), pp. 312–319.

[RVO07] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. "CArtAgO: A Framework for Prototyping Artifact-Based Environments in MAS". In: *Environments for MultiAgent Systems III. 3rd International Workshop, E4MAS 2006, Hakodate, Japan, May 8, 2006, Selected Revised and Invited Papers* 4389 (2007), pp. 67–86.

[SST13] Mohammad Shaker, Noor Shaker, and Julian Togelius. "Evolving playable content for cut the rope through a simulation-based approach". In: *Artificial Intelligence and Interactive Digital Entertainment* (2013), pp. 72–78.

[Syc98] Katia P Sycara. "Multiagent Systems". In: 19.2 (1998), pp. 79–92.

[VB91] J. David Velleman and Michael E. Bratman. *Intention, Plans, and Practical Reason.* Vol. 100. 2. 1991, p. 277.

[Vla03] Nikos Vlassis. "A Concise Introduction to Multiagent Systems and Distributed AI". In: *University of Amsterdam, Amsterdam* (2003), pp. 1–76.

[Woo09] Michael Wooldridge. *An introduction to multiagent systems*. 2009.

[Uni17] Unity Technologies. *Unity Execution Order*. 2017. URL: https://docs.unity3d.com/Manual/ExecutionOrder.html.

[Uni18] Unity Technologies. *Unity3D*. 2018. URL: https://unity3d.com/.